An Empirical Study on Fine-tuning Large Language Models of Code for Automated Program Repair

Kai Huang*†‖, Xiangxin Meng‡, Jian Zhang§¶, Yang Liu§, Wenjie Wang*, Shuhao Li†, Yuqing Zhang*†¶
*University of Chinese Academy of Sciences, Beijing, China
†Zhongguancun Laboratory, Beijing, China
‡Beihang University, Beijing, China
§Nanyang Technological University, Singapore
huangk@nipc.org.cn, mengxx@act.buaa.edu.cn, jian_zhang@ntu.edu.sg, yangliu@ntu.edu.sg,
wangwj@ucas.ac.cn, lishuhao@zgclab.edu.cn, zhangyq@nipc.org.cn

***Abstract*—The advent of large language models (LLMs) has opened up new opportunities for automated program repair (APR). In particular, some recent studies have explored how to leverage large language models of code (LLMCs) for program repair tasks and show promising results. However, most of them adopt the zero/few-shot learning paradigm for APR, which directly use LLMCs to generate the possibly correct code given its surrounding context. Though effective, the repair capabilities of LLMCs based on the fine-tuning paradigm have yet to be extensively explored. Also, it remains unknown whether LLMCs have the potential to repair more complicated bugs (e.g., multi-hunk bugs). To fill the gap, in this work, we conduct a comprehensive study on the program repair capability of LLMCs in the fine-tuning paradigm. We select 5 popular LLMCs with representative pre-training architectures, including CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder. We consider 3 typical program repair scenarios (i.e., bugs, vulnerabilities, and errors) involving 3 programming languages (i.e., Java, C/C++, and JavaScript). Notably, we take both single-hunk and multi-hunk bugs/vulnerabilities into account. We then fine-tune them on widely-used datasets and compare them with existing state-of-the-art APR tools. We also investigate the impact of different design choices, which include code abstractions, code representations, and model evaluation metrics. Our experimental results show that LLMCs in the fine-tuning paradigm can significantly outperform previous state-of-the-art APR tools. Through in-depth analysis, we provide insights into choosing appropriate strategies to guide LLMCs for better performance. Lastly, we reveal several limitations of LLMCs for APR and make suggestions for future research on LLMC-based APR.**

***Index Terms*—Automated Program Repair, Large Language Models of Code, Neural Machine Translation, Fine-Tuning**

I. Introduction

Automated program repair (APR) techniques [1]–[6] aim to automate the repair of software defects to reduce manual work and guarantee the software quality. Among them, learning-based APR techniques [5], [6] have attracted much attention in recent years. In general, the Neural Machine Translation (NMT) model is adopted for supervised training on bug-fix pairs (BFPs) [7], which translates the buggy program to its fixed version. Compared to traditional APR techniques [1]–[3], both the quantity and diversity of fixed bugs have been improved through the use of learning-based tools [5], [8], [9].

Existing works [9]–[15] mainly employ traditional neural models (e.g., RNN, Transformer, etc.) in an encoder-decoder framework and incorporate different kinds of domain knowledge to automatically learn source code semantics for repairing buggy programs. For example, DLFix [11] uses a tree-based RNN model that learns the contexts of bug fixes, resulting in an additional weighting input for learning the bug-fixing code transformations. Similarly, Recoder [9] proposes a syntax-guided edit decoder for APR with a provider/decider architecture to predict syntactically correct edits. It uses placeholders to generate patches with project-specific identifiers. Nevertheless, the scale of model parameters along with the training data is limited, which may fail to learn more strict syntactic features and complex semantic dependencies of code elements, thus may not generalize to unseen bug types [8], [16].

In contrast, large language models of code (LLMCs) have shown great advantages over traditional NMT-based models of APR [17]–[19]. For example, Xia et al. [17] propose AlphaRepair, a bug repair tool based on the LLMC, which uses CodeBERT [20] as the foundation model and uses the zero-shot learning paradigm to leverage the LLMC's own code understanding and generation capabilities for bug repair tasks. Recently, researchers are becoming more aware of LLMCs and have conducted studies to evaluate its impact on APR [16], [21]–[23]. These studies indicate that directly applying pre-trained LLMs can already substantially outperform all existing APR tools. For instance, the InCoder model can fix 72% more bugs than traditional learning-based APR techniques [22].

Despite all this, most existing studies focus on directly using the LLMC with zero-shot or few-shot prompting, while the benefit of fine-tuning LLMCs for APR is not yet fully understood [6]. The missed opportunities are twofold: 1) Even though Jiang et al. [22] explored the LLMC fine-tuning, however, as stated, the applied fine-tuning is straightforward and simple. It is still unclear regarding the impact of specific design choices with fine-tuning, and what factors limit the repair capability of LLMCs. 2) The scope of prior work typically lies within limited bug types (e.g., single-hunk and

‖Work done while this author was an intern at Nanyang Technological University, Singapore.
¶Corresponding authors.

common bugs), the generality of APR tools has been largely neglected (e.g., multi-hunk bugs and vulnerabilities).

To address the above issues, this paper comprehensively explores the repair ability of LLMCs under the NMT fine-tuning paradigm, aiming to provide more empirical guidance for APR research and further bridge the gap between LLMCs and APR. We investigate the repair capability of 5 LLMCs in the NMT fine-tuning paradigm by following an encoder-decoder generation procedure on bug-fix pairs (BFPs). We apply the fine-tuned LLMCs on 7 popular evaluation benchmarks of different programming languages (PLs), and take into account software bugs, security vulnerabilities, and programming errors. For simplicity, we collectively refer to them as *defects*. Apart from single-hunk defects, we also evaluate the ability of LLMCs for repairing multi-hunk defects. Specifically, we focus on the following key aspects when using LLMCs for APR.

Repair Effectiveness. We study the repair effectiveness of LLMCs under the NMT fine-tuning paradigm in 6 repair tasks. Our results show that LLMCs' repair capability outperforms previous APR tools. For bug repair, the best model fixes 34 and 25 bugs more than Jiang et al. [22] and Li et al. [24]; For vulnerability repair, the best repair accuracy is improved by 20.04% compared to VulRepair [18]. For error repair, our best repair accuracy outperforms TFix [25] by 11.32%. Interestingly, we find that the smaller model (UniXcoder) matched or even surpassed the larger model (CodeT5) in terms of the repair capability. This indicates it does not necessarily select LLMCs as large as possible for program repair.

Design Choices. We undertake an in-depth exploration of different design choices in fine-tuning. We consider three typical kinds of strategies, including code abstraction, code representation, and checkpoint selection. The results imply that: 1) The code abstraction strategy used in earlier work [7] is unsuitable for LLMCs and may even reduce the repair capability of LLMCs. 2) Using the representation of buggy code with fault locations and fix code without surrounding tokens can yield better repair results. 3) Different repair scenarios may require different evaluation metrics for checkpoint selection to obtain the best fine-tuned model. And we find the ensemble strategy that combines multiple selected checkpoints is a good way to enhance the repair effectiveness.

Multi-hunk Defects. Unlike previous studies [16], [22], for the first time, we thoroughly examine the performance of NMT fine-tuned LLMCs for repairing multi-hunk defects. To realize that, we adopt a code representation to mark the fault locations and their fixes, which can learn to generate fixed code for the multiple hunks correspondingly. Here, we explore the multi-hunk bug/vulnerability repair. Overall, the best LLMC fixed 9 more multi-hunk bugs than the dedicated multi-hunk APR tool DEAR [24]. Surprisingly, LLMCs exhibit similar performance between single-hunk (except single-line) and multi-hunk bug/vulnerability repair. While for overly complex vulnerabilities (e.g., more than 5 hunks), the repair accuracy decreases dramatically with the increase of hunks. Nevertheless, LLMCs can still fix a small portion (i.e., 5%) of them even when the number of hunks exceeds 10. These findings may encourage researchers to pay more attention to complex defects for more practical use.

Future Directions. The limitations and challenges of fine-tuning LLMCs for APR are also analyzed. We thoroughly discussed two major factors that affect the repair capability of LLMCs. On one hand, LLMCs usually fail to generate correct patches due to lack of repair ingredients (e.g., method name of one class), which may blame to the input/output length limit of LLMCs. On the other hand, the generated candidate patches are greatly restricted, since the hardware resources often cannot meet the high computational demands of large models for massive patches. Based on this, we propose some mitigation measures and future directions (Section VII). For example, we can adopt a sliding-encoder and decoder (SLED) [26] on the pre-trained models to accept long and/or dependent methods. Besides, sparse mechanism [27] and limited discrepancy beam search [28] can also be applied when generating larger number of candidate patches.

To sum up, this paper makes the following contributions:

- We systematically study the capability of 5 representative LLMCs (CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder) in the fine-tuning paradigm for APR across 3 typical repair scenarios (software bugs, security vulnerabilities, and programming errors).
- We conduct an in-depth investigation of design choices that may enhance the repair capability, and achieve significant improvements over vanilla fine-tuning using same models.
- Our study reveals that LLMCs are capable of repairing fairly complex multi-hunk defects to some extent, and there is a good potential to tackle more complicated ones.
- We analyze several key factors that limit the repair capability of LLMCs and propose feasible solutions to mitigate them.
- Our artifacts including the code and experimental data are made publicly available [29], which can serve as benchmarks and baselines for the future work.

II. Methodology

A. Large Language Model of Code

LLMCs can be divided into three architectures: encoder-only, decoder-only, and encoder-decoder [16], [22], [30], [31]. **1) Encoder-only** LLMCs are pre-trained based on BERT using Mask Language Model (MLM), including CodeBERT [20], GraphCodeBERT [32], ContraBERT [33], etc. **2) Decoder-only** LLMCs are represented by GPT, which has only the decoder and pre-trained in an autoregressive manner. Typical models are CodeGPT [34], GPT-C [35], Codex [36] etc. **3) Encoder-Decoder** LLMCs retain the infrastructure of Transformer with both encoder and decoder, such as CodeT5 [37] and PLBART [38].

B. Overall Workflow

The workflow of fine-tuning LLMCs for APR follows the basic learning-based APR technique [5], as shown in Figure 1.

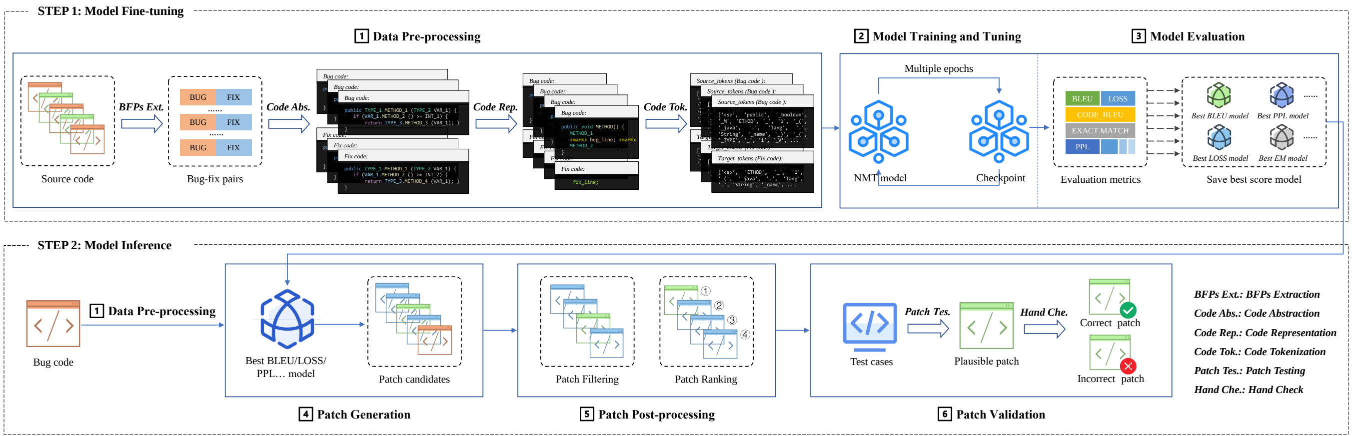

Fig. 1: The workflow for APR based on NMT fine-tuning of LLMCs.

Generally, applying LLMCs to the APR workflow in the NMT fine-tuning paradigm involves the following steps: 1) data pre-processing, 2) model training and tuning, 3) model evaluation, 4) patch generation, 5) patch post-processing, and 6) patch validation. Next, we describe the technical details of each step.

C. Data Pre-processing

Data pre-processing phase aims to convert the raw source code into a format that LLMC can efficiently process. We adopt the common practice of using BFPs [7] for learning to transform the buggy code to fixed code at the method level.

1) Code Abstraction: Code abstraction processing was first introduced to bug repair tasks by Tufano et al. [7]. This technique alleviates the out-of-vocabulary (OOV) problem by normalizing code elements and facilitates models to learn generic fixing patterns [5], [6]. Subsequently, many following works [10], [11], [24], [39], [40] adopt the similar strategy for improvement. However, it is unclear whether code abstraction could benefit LLMCs. Therefore, we explore the impact of code abstraction [7] as the first design choice.

2) Code Representation: In learning-based APR techniques, code representation is an essential factor for the repair capability. Earlier works only focused on single-hunk bugs and design the code representation specific to them. Recently, VRepair [41] has extended the NMT model to multi-hunk fixes by improving the code representation. To explore the impact of different code representations on LLMCs' repair capability, we consider four code representations, abbreviated as *CR1*, *CR2*, *CR3*, and *CR4*. As shown in Figure 2, all of them are based on token sequence because existing LLMCs are generally limited to such a representation. The details are illustrated below.

Fig. 2: Four code representation forms.

CR1: This is the original representation of NMT-based APR work [7], which takes a whole buggy method as input and a whole fixed method as output. *CR1* aims to allow the model to automatically fix defects without fault localization (FL).

CR2: *CR2* is based on CR1, where the bug/fix hunk are marked with special tokens (<BUGS>, <BUGE>, <FIXS>, <FIXE>) so that the model learns the transition from bug code to fixed code with the help of FL information. Hence, we use it to analyze the impact of FL information on the repair capability.

CR3: Inspired by SequenceR [10], we remove the context of fixed code from *CR2* to reduce the model output length and speed up the training and prediction. This representation is used to analyze the impact of simplifying the learning target (i.e., the output) on the repair capability.

CR4: This is VRepair's code representation for multi-hunk fixes [41]. Unlike *CR3*, *CR4* uses different mark ways to distinguish between different repair behaviors (i.e., add, delete, replace) and therefore has a finer marker granularity. Through comparison, we can analyze the impact of fine-grained representation on repair capability.

Note that the above four code representations support both single-hunk and multi-hunk repair scenarios.

3) Code Tokenization: Following previous works [8], [18], [25], [39], [42], we use a subword-level tokenizer namely byte-pair encoding (BPE). It replaces frequently occurring sequences of characters with a single symbol, resulting in a more compact vocabulary. Therefore, it can effectively alleviate the OOV problem in APR [5], [6] and is superior to the word-level tokenizer [18].

D. Model Training and Tuning

This step aims to extend LLMCs into the NMT model architecture for fine-tuning. For encoder-only LLMCs, we add decoders to build the Seq2Seq architecture and fine-tune them in a supervised manner. For encoder-decoder LLMCs, they are the Seq2Seq architecture, so no changes to the structure are needed. However, for decoder-only LLMCs, such generative models need to concatenate the input and output for fine-tuning, which weakens the ability of understanding buggy code semantics due to the length limit. And a recent study [31] indicates that decoder-only LLMCs can perform significantly worse than the above two kinds of LLMCs. Therefore, we focus on encoder-only and encoder-decoder LLMCs in this work. After building the NMT model, multiple training iterations are performed on the training dataset to enable the model to learn the domain knowledge for defect repair.

E. Model Evaluation

During model training and tuning, the performance of checkpoints after each training round need to be evaluated on the validation set to find the best trained model. Researchers have proposed various metrics for model evaluation [43], such as *PPL*, *BLEU*, etc. However, it is still unclear how they can affect the selection of the best repair model. Therefore, we explore them to guide the selection of checkpoints in APR tasks. Besides, we also keep the last round of checkpoints that is irrelevant evaluation metrics, which we call the *Last model*.

F. Patch Generation and Validation

In the patch generation phase, we use the beam search strategy on multiple repair models from the model evaluation phase to perform patch synthesis. We do not consider the post-processing since it has already been well studied [16] and is out of our scope. In the benchmark with test cases, we followed the validation strategy from previous works [8]–[12], [14], [17], [24], [42], [44], [45]. First, we run test cases to filter out plausible patches. Then two authors manually check the plausible patches to determine whether a plausible patch is correct or incorrect patch. Finally, the result is *correct patches / plausible patches* (X/Y). In the benchmark without test cases, we follow previous works [7], [18], [25], [39]–[41] and use the *exact match* strategy to calculate the *repair accuracy* (Z%).

TABLE I: Details of the selected LLMCs.

Model	Size	Type	Dataset
CodeBERT	125M	Encoder	CodeSearchNet
GraphCodeBERT	125M	Encoder	CodeSearchNet
PLBART	140M	Encoder-Decoder	StackOverflow and BigQuery
CodeT5	220M	Encoder-Decoder	CodeSearchNet and BigQuery
UniXcoder	125M	Encoder-Decoder	CodeSearchNet

III. Experimental Setup

A. Research Questions

We explore the repair capability of LLMCs in different scenarios by answering the following research questions in software bug repair, security vulnerability repair, and programming error repair, respectively:

RQ1: How do different design choices affect LLMCs' repair capability? RQ1 investigates the impact of different design choices on LLMCs' repair capability, which can help better compare LLMCs and provide guidance on fine-tuning LLMCs. We will explore the impact of code abstraction (Section II-C1), code representation (Section II-C2), and checkpoint selection (Section II-E) on the results in the experiment.

RQ2: How well does the LLMC perform compared to the state-of-the-art approaches? RQ2 aims to explore the repair capability of LLMCs. We systematically evaluate their performance under multiple defect types, programming languages, and defect complexities. Further, we compare LLMCs to SOTA APR works to know whether LLMCs are superior.

RQ3: What are the factors that limit the effectiveness of fine-tuning LLMCs? RQ3 aims to reveal the shortcomings of LLMCs for APR tasks when fine-tuning and point out some future directions for improvement.

B. Studied LLMCs

We follow the following criteria for selecting LLMCs. First, we assume that the computing resource should be readily available (e.g., a RTX 3090 GPU), which means that the model size is at the million level. Also, the model parameters of all LLMCs should be of similar size for a fairer comparison of the repair capability. Second, the pre-trained model and its data should be open-sourced, which allows for fine-tuning models and analyzing the pre-training data (e.g., data leak).

Finally, we choose 5 models: CodeBERT [20], GraphCode-BERT (GraphCode) [32], PLBART [38], CodeT5 [37], and UniXcoder [46]. More details of models are shown in Table I.

C. Datasets and Baselines

As shown in Table II, we describe the datasets and baselines.

1) Software Bug Repair: In bug repair tasks, we evaluate LLMCs on large-scale datasets and small-scale benchmarks.

Datasets. We use the **BFP dataset** including the small and medium versions (BFP_S and BFP_M) provided by Tufanol et al. [7] (Task ❶) and the **SequenceR dataset** (SeqRD) provided by Chen et al. [10] (Task ❷) for training and testing, and take their approaches as baselines.

Benchmarks. We use the standard **Defects4J** (D4J) [47] as the test benchmark. For the single-hunk bug repair (Task ❸), we use the **Recoder dataset** (RecD) provided by Jiang et

TABLE II: Datasets, baselines, and parameter settings for experimental setups. (I.O.: Max Input/Output Length; L.R.: Learning Rate; T.E.: Training Epoch; B.S.: Beam Size; P.N.: Patch Number)

Repair Task		Training Dataset			Test Benchmark			Language	Defect Complexity		Baseline	Parameter Setting				
Defect	Task	Dataset	#Bugs	#BFPs	Dataset	#Bugs	#BFPs		Single-hunk	Multi-hunk		I.O.	L.R.	T.E.	B.S.	P.N.
Bug	❶	BFP_S	-	52,515	BFP_S	-	5,835	Java	✓	✓	Tufano et al. [7]	512	5e-5	30	5	1
		BFP_M	-	58,909	BFP_M	-	6,546	Java	✓	✓	Tufano et al. [7]	512	5e-5	30	5	1
	❷	SeqRD	-	35,551	SeqRD	-	4,707	Java	✓		Chen et al. [10]	512	5e-5	30	50	50
	❸	RecD	-	143,666	D4J V1.2	383	554	Java	✓		Jiang et al. [22]	512	5e-5	30	100	10
		RecD	-	143,666	D4J V2.0	412	719	Java	✓		Jiang et al. [22]	512	5e-5	30	100	10
	❹	CPMD	44,154	80,501	D4J V1.2	383	554	Java	✓	✓	Li et al. [24]	512	1e-4	30	100	100
Vul.	❺	VulRD	-	6,776	VulRD	-	1,706	C/C++	✓	✓	Fu et al. [18]	512	2e-5	75	50	50
Error	❻	TFixD	-	94,300	TFixD	-	10,504	JavaScript	✓		Berabi et al. [25]	512	2e-5	30	5	1

al. [22] for model training, and the testing is on Defects4J V1.2 and 2.0. For the multi-hunk bug repair (Task ❹), we use the **CPatMiner dataset** (CPMD) provided by Li et al. [24] for training, and test them on the Defects4J V1.2. As Li et al. only provided repair results on Defects4J V1.2, we keep consistent with them to avoid bias on V2.0. Note that we extracted 220,125 BFPs from the CPatMiner dataset and ended up with 80,501 BFPs after removing duplicate ones. Besides, Defects4J V1.2 contained 395 bugs, and Defects4J V2.0 introduced additional 444 bugs. Since we focus on method-level bug fixing, we use 383 and 412 bugs of them.

2) Security Vulnerability Repair: We use VulRepair by Fu et al. [18] as the baseline and their **VulRepair dataset** (VulRD) for model training and testing (Task ❺), which include Big-Vul [48] and CVEfixes [49].

3) Programming Error Repair: We use TFix by Berabi et al. [25] as the baseline and their publicly available **TFix dataset** (TFixD) for training and testing (Task ❻).

D. Implementation

The parameter settings for each repair task are listed in Table II. Besides, we use perfect fault localization (PFL) and set a maximum run time limit of 5 hours [9], [24], [50]. All the pre-trained models are downloaded from Hugging Face. We conduct all the experiments on a 12-Core workstation with Intel(R) Xeon(R) Bronze 3204 CPU, 46 GB RAM and 24G RTX3090 GPU, running Ubuntu 18.04.6 LTS.

IV. Software Bug Repair

A. Empirical Results

Table III shows the repair results of LLMCs in different repair tasks: **1) In Task ❶**, we first study the impact of different design choices for bug repair on the BFP dataset [7] to provide guidance for subsequent experiments. These choices include different code abstraction strategies (*abs*/*raw*), code representation forms (*CR1*/*CR2*/*CR3*), and model evaluation metrics for selecting checkpoints (*PPL*/*BLEU*/*Last*). **2) In Task ❷**, we follow the insights gained from Task ❶ and use the best combination of the code representation (*CR3*) + without code abstraction (*raw*) to perform the experiments on the SequenceR dataset [10]. **3) In Task ❸**, we evaluate LLMCs for single-hunk bugs on Defects4J V1.2 and V2.0 [47]. **4) In Task ❹**, we evaluate the multi-hunk bugs on Defects4J V1.2.

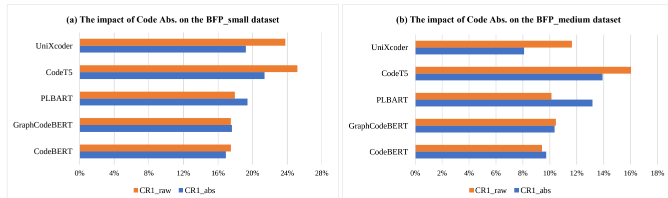

Fig. 3: The impact of Code Abs. on the BFP dataset.

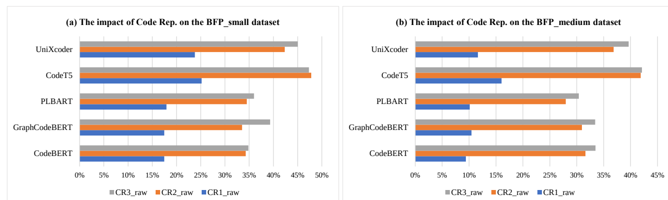

Fig. 4: The impact of Code Rep. on the BFP dataset.

B. Research Questions

1) RQ1: **How do different design choices affect LLMCs' repair capability?**

Code Abstraction. We first analyze the impact of code abstraction. From Table III (Task ❶), we observe that using code abstraction (*CR1_abs*) and without code abstraction (*CR1_raw*) have a more or less impact on the repair results. To present a more intuitive picture of the impact of code abstraction, we extract the *Last model* repair results for each LLMC for comparison (*Last model* is irrelevant to evaluation metrics). As shown in Figure 3, for most LLMCs, the potential impact of code abstraction on the repair capability may be limited or negligible. For example, CodeT5 and UniXcoder achieved the best repair results using *raw* code input, while CodeBERT and GraphCodeBERT have close results on *abs* and *raw*. This suggests that *raw* code is already adequate, and it is unnecessary to use code abstraction for LLMCs.

There are two main reasons for the phenomenon. First, since LLMCs are usually pre-trained on *raw* source code, they are better suited to the same unprocessed *raw* data for downstream tasks. Second, as Chen et al. argue, code abstraction may lose some semantic information (e.g., special function and variable names) [10], which makes it difficult to learn fix patterns.

Finding 1: Code abstraction does not significantly improve the repair capability of most LLMCs. Using the data format without code abstraction processing (i.e., *raw* source code) is more suitable for fine-tuning LLMCs.

Code Representation. In order to investigate the impact

TABLE III: Repair results of LLMCs in different repair tasks. (X/Y: correct patches / plausible patches; Z%: repair accuracy)

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❶	BFP_S	CR1_abs	12.94%	**16.90%**	**16.90%**	13.32%	17.55%	**17.62%**	17.96%	8.86%	**19.40%**	20.15%	**21.80%**	21.37%	**19.47%**	18.89%	19.20%
		CR1_raw	12.17%	16.30%	**17.48%**	14.81%	17.26%	**17.46%**	14.41%	17.86%	**17.93%**	19.69%	22.45%	**25.18%**	18.44%	**23.79%**	**23.79%**
		CR2_raw	**34.28%**	**34.28%**	**34.28%**	31.11%	33.44%	**33.52%**	31.65%	34.14%	**34.52%**	43.27%	**47.82%**	47.80%	36.71%	41.70%	**42.33%**
		CR3_raw	34.82%	**35.25%**	34.82%	36.49%	**39.32%**	**39.32%**	33.13%	34.57%	**36.00%**	42.71%	**47.66%**	47.34%	42.67%	**45.02%**	**45.02%**

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❶	BFP_M	CR1_abs	4.72%	3.16%	**9.73%**	3.96%	3.96%	**10.36%**	**13.17%**	0.31%	**13.17%**	8.27%	**13.95%**	13.92%	5.39%	2.55%	**8.08%**
		CR1_raw	4.22%	**9.41%**	**9.41%**	5.27%	**10.45%**	**10.45%**	4.58%	0.69%	**10.13%**	8.72%	**16.03%**	**16.03%**	6.59%	11.38%	**11.64%**
		CR2_raw	26.11%	**31.75%**	31.61%	27.56%	30.88%	**30.96%**	23.47%	**27.96%**	**27.96%**	35.77%	**41.88%**	**41.88%**	32.21%	**36.85%**	**36.85%**
		CR3_raw	28.40%	**33.48%**	**33.48%**	29.01%	33.35%	**33.43%**	27.04%	30.39%	**30.74%**	36.24%	**42.12%**	**42.12%**	36.29%	39.53%	**39.65%**

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❷	SeqRD	CR3_raw	**19.06%**	14.40%	14.49%	**19.35%**	14.74%	14.53%	**17.87%**	13.13%	12.96%	**36.22%**	26.20%	26.03%	**33.91%**	22.33%	22.24%

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❸	D4J V1.2	CR3_raw	21/37	**25/35**	23/31	24/39	26/35	**26/37**	12/24	**16/23**	**16/23**	**41/53**	30/43	30/43	**46/63**	36/46	38/45
	D4J V2.0	CR3_raw	22/38	12/26	**23/26**	**23/43**	22/38	22/37	**23/39**	15/27	15/27	**30/39**	16/26	16/26	**37/55**	20/33	22/35
	D4J V1.2	CR3_raw	**7.76%**	6.68%	6.14%	**8.84%**	6.86%	7.04%	4.51%	**4.69%**	**4.69%**	**11.01%**	8.84%	8.84%	**11.73%**	8.48%	9.21%
	D4J V2.0	CR3_raw	**5.84%**	2.78%	3.20%	**5.70%**	4.03%	4.03%	**6.12%**	4.59%	4.59%	**8.34%**	6.12%	6.12%	**8.84%**	5.70%	5.84%

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❹	D4J V1.2	CR3_raw	**40/54**	34/48	31/46	**41/61**	39/53	39/53	**32/52**	25/39	25/41	**69/95**	53/77	53/77	**66/102**	52/76	54/82
	D4J V1.2	CR3_raw	**11.19%**	10.47%	10.83%	**10.11%**	8.84%	8.84%	**10.29%**	7.94%	7.76%	**17.69%**	12.64%	12.64%	**18.41%**	13.36%	14.08%

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❺	VulRD	CR3_raw	43.96%	**51.58%**	**51.58%**	43.61%	**53.28%**	53.22%	47.30%	59.85%	**60.08%**	52.93%	**62.78%**	**62.78%**	50.64%	62.08%	**62.19%**
		CR4_raw	31.07%	49.00%	**49.24%**	25.91%	41.85%	**41.97%**	46.19%	53.58%	**53.93%**	35.87%	55.86%	**55.92%**	40.62%	55.28%	**55.63%**

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❻	TFixD	CR3_raw	48.28%	52.39%	**52.83%**	48.53%	**52.33%**	**52.33%**	44.78%	**48.23%**	**48.23%**	50.44%	54.93%	**55.31%**	48.25%	**54.33%**	54.31%

of different code representations, we again compare LLMCs with the *Last model* according to the results from Table III (Task ❶). We present the impact of the different CRs on the repair results in Figure 4. As shown in Figure 4, the use of *CR2* with fault location and repair location information outperforms the repair results of *CR1*. In addition, we observe that *CR3*, a representation that removes the repair code context information, has a slightly better repair effect than *CR2*. Combining these results, we conclude that *CR3* is a more suitable code representation for the repair task.

We now analyze why *CR3* can work more effectively on LLMCs. First, using special tokens to mark fault/repair locations enables the model to focus on code repair behaviors for targeted learning. Second, LLMCs suffer from the long sequence problem [18]. As the length of the input/output grows, the repair accuracy of LLMCs decreases. Removing irrelevant context from the output sequence is equivalent to reducing the output length, so the model's repair capability may be improved.

Finding 2: The fine-tuning of LLMCs for APR can be improved by delicate code representations. More accurate fault location, precise repair location information, and removing the contextual code of the fixes are all beneficial for improvement.

Checkpoint Selection. We track the impact of different model evaluation metrics for checkpoint selection on the results under the best input/output format *CR3_raw*. In Table III (Task ❶), on the BFP dataset, the *best BLEU model* and the *Last model* tend to achieve higher repair accuracy than the *best PPL model*. However, the contrary result is obtained on the SequenceR dataset and Defects4J. In Table III (Task ❷-❹), the *best PPL model* achieves the best repair accuracy.

This motivates us to explore further. First, we noticed that in the repair scenario with *BLEU* as the best metric (Task ❶), the train/val/test datasets were obtained from a random split on the BFP dataset. Thus the train/val/test datasets hold similar data characteristics. Second, we found that in the repair scenarios where *PPL* was the best metric (Task ❷-❹), the train/val/test datasets were not split from the same dataset. For example, on Task ❷, the training data of the SequenceR dataset consists of CodRep 1/2/3/5 [51] and the BFP dataset [7], whereas the test data comes from CodRep 4 [51]. Similarly, on Task ❸ and Task ❹, the training data (Recoder dataset and CPatMiner dataset) do not contain the software projects from Defects4J. We can conclude that *BLEU* aligns better with training data, while *PPL* shows better generalization. When data characteristics are alike, *BLEU* is better; otherwise, *PPL* is preferred. However, it is hard to know the difference between the training and testing samples in practice. Therefore, we follow the practice of previous works [12], [52] to use the ensemble strategy by combining multiple checkpoints (*PPL*/*BLEU*/*Last*) to enhance the repair capability.

TABLE IV: Best repair results for LLMCs on Task ❶

Benchmark	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	Tufano et al.
BFP_S	36.59%	44.44%	43.62%	54.14%	51.48%	10.32%
BFP_M	37.74%	38.04%	36.59%	47.49%	46.26%	3.61%

TABLE V: Best repair results for LLMCs on Task ❷

CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	SequenceR
21.92%	22.46%	20.31%	38.11%	36.63%	20.18%

Finding 3: Different repair scenarios may have different best evaluation metrics for checkpoint selection. In practice, using ensemble learning is an appropriate strategy.

2) RQ2: **How well does the LLMC perform compared to the state-of-the-art approaches?** Based on findings obtained from RQ1, we use the *CR3_raw* and the ensemble strategy to obtain the best performance of LLMCs and compare them with baselines.

Task ❶. As shown in Table IV, on the *BFP dataset*, LLMCs improve over the baseline Tufano et al. [7] as follows: 1) BFP_S: CodeT5 (+43.82%) > UniXcoder (+41.16%) >

TABLE VI: Best repair results for LLMCs on Task ❸

Benchmark	Our Work					Baseline (Jiang et al. [22])										
	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	PLBART	CodeT5	CodeGen			InCoder		DL-based APR Tool			
	base	base	base	base	base	base	base	350M	2B	6B	1B	6B	CURE	Reward	Recoder	KNOD
D4J V1.2	33/47	34/47	16/31	49/62	**57/71**	25/-	30/-	23/-	32/-	38/-	27/-	41/-	6/-	20/-	24/-	20/-
D4J V2.0	25/44	28/48	24/41	33/45	**46/67**	13/-	17/-	20/-	23/-	23/-	24/-	28/-	6/-	8/-	11/-	13/-

TABLE VII: Best repair results for LLMCs on Task ❹

Bug Type	Our Work					Baseline (Li et al. [24])			
	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	DEAR	CURE	CoCoNut	DLFix
1. One-Hunk of One-Stmt	21/27	24/31	16/25	42/54	42/52	33/-	38/-	37/-	35/-
2. One-Hunk of Multi-Stmts	8/10	7/11	7/9	11/14	11/16	4/-	3/-	3/-	1/-
3. Multi-Hunks of One-Stmt	10/11	9/14	9/13	12/16	11/18	13/-	6/-	3/-	4/-
4. Multi-Hunks of Multi-Stmts	6/6	4/5	4/4	5/5	6/8	1/-	0/-	0/-	0/-
5. Multi-Hunks of Mix-Stmts	7/8	3/6	3/7	7/17	8/22	2/-	1/-	1/-	0/-
Total	**52/62**	**47/67**	**39/58**	**77/106**	**78/117**	**53/-**	**48/-**	**44/-**	**40/-**

GraphCodeBERT (+34.12%) > PLBART (+33.30%) > CodeBERT (+26.27%). 2) BFP_M: CodeT5 (+43.88%) > UniXcoder (+42.65%) > GraphCodeBERT (+34.43%) > CodeBERT (+34.13%) > PLBART (+32.98%).

Task ❷. On the *SequenceR dataset*, all LLMCs' results outperform SequenceR [10] (Table V): CodeT5 (+17.93%) > UniXcoder (+16.45%) > GraphCodeBERT (+2.28%) > CodeBERT (+1.74%) > PLBART (+0.13%).

Task ❸. As shown in Table VI, our best results on Defects4J outperform previous APR tools [8], [9], [14], [50] and the recent study [22]. On Defects4J V1.2 and V2.0, our CodeT5-base and PLBART-base fixed 35 and 2 more bugs than Jiang et al. [22] when using the same model fine-tuning. Notably, our small-scale LLMCs UniXcoder/CodeT5 outperform large-scale LLMC InCoder-6B used by Jiang et al. Compared to their best model InCoder-6B, our results are as follows: 1) Defects4J V1.2: UniXcoder (+16) > CodeT5 (+8) > GraphCodeBERT (-7) > CodeBERT (-8) > PLBART (-25). 2) Defects4J V2.0: UniXcoder (+18) > CodeT5 (+5) > GraphCodeBERT (+0) > CodeBERT (-3) > PLBART (-4).

Task ❹. As shown in Table VII, our best repair results on Defects4J V1.2 outperform existing works [8], [11], [12], [24]. LLMCs improve over the best multi-hunk APR tool DEAR [24] as follows: UniXcoder (+25) > CodeT5 (+24) > CodeBERT (-1) > GraphCodeBERT (-6) > PLBART (-14).

According to the results from Task ❶-❹, we find that using the LLMCs UniXcoder and CodeT5 has surpassed previous works on bug repair tasks. This demonstrates that fine-tuning LLMC has great potential for APR research. Notably, the smaller UniXcoder-base (125M) achieved similar or even better results than CodeT5-base (220M).

> **Finding 4**: The repair capability of LLMCs show great potential for bug repair tasks. In addition, small-scale models may achieve similar or even better results than larger models.

Multi-Hunk. We also pay close attention to multi-hunk bug fix, since little research has been done on the repair of complex bugs. As mentioned earlier, we use *CR3* to extend the NMT workflow to multi-hunk repair scenarios. The results are provided in Table VII. Obviously, compared to single-line bugs (Type 1), single-hunk (Type 2) and multi-hunk bugs (Type 3-5) are far more difficult to repair. This is because such bugs entail intricate dependencies from both inner and outer of one buggy method. Nonetheless, our work achieves a great improvement over existing approaches on fixing such complex bugs. In particular, UniXcoder and CodeT5 outperform the advanced multi-hunk APR tool DEAR and fixed 9 and 8 more multi-hunk bugs. Furthermore, to our surprise, there is little gap between the number of fixed single-hunk bugs and the number of multi-hunk bugs. We hope such results can encourage researchers to explore more advanced approaches for repairing complex bugs, as they are more common in real-world projects.

> **Finding 5**: Fine-tuning LLMCs to fix multi-hunk bugs is also promising, though it is more difficult compared with single-line bug repair.

Data Leakage. Prior work AlphaRepair [17] uncovered the data leak issue when using LLMCs for APR, that is, the overlap between the pre-training data (CSN, i.e.,CodeSearchNet) and the test benchmark (D4J, i.e.,Defects4J). To reveal its impact, we follow AlphaRepair to analyze the LLMCs' repair results with respect to the data leak. We searched for the exact match and identified 48 overlaps between D4J IDs and CSN. Then we checked if the patches generated by LLMCs were identical to those present in D4J. In Task ❸, among the 48 patches, only Lang_43, Math_22, and Mockito_5 were identified, and a similar pattern was observed in Task ❹. The results imply that fine-tuned LLMCs are minimally affected by data leakage. However, this also means that LLMCs may tend to lose certain pre-trained knowledge during the fine-tuning process, highlighting the presence of the catastrophic forgetting problem [53], [54] during this phase.

> **Finding 6**: The LLMC is minimally affected by data leakage after sufficient fine-tuning. However, this exposes the catastrophic forgetting problem of LLMCs under the fine-tuning paradigm.

3) RQ3: **What are the factors that limit the effectiveness of fine-tuning LLMCs?** We found two main factors that may limit the repair capability of LLMCs in our experiment.

Lack of Repair Ingredients. One of the factors is the method-level BFPs and the limited input/output length. In particular, when using method-level BFPs, it is difficult for the model to synthesize correct patches based on the incomplete context if repair ingredients (e.g., method names, variable names, etc.) are outside of that method. Besides, if a method exceeds the maximum input/output length, it is difficult to provide a complete method for the model. This may result in

TABLE VIII: Results of different design choices on Task ❺

Model + Design Choices	Repair Accuracy
CodeT5 + CR4_raw + LOSS (VulRepair)	44.67%
CodeT5 + CR4_raw + PPL	35.87% (-8.8%)
CodeT5 + CR4_raw + BLEU	55.86% (+11.19%)
CodeT5 + CR4_raw + Last	55.92% (+11.25%)
CodeT5 + CR4_raw + Ensemble Learning	57.33% (+12.66%)
CodeT5 + CR3_raw + LOSS	52.35% (+7.68%)
CodeT5 + CR3_raw + PPL	52.93% (+8.26%)
CodeT5 + CR3_raw + BLEU	62.78% (+18.11%)
CodeT5 + CR3_raw + Last	62.78% (+18.11%)
CodeT5 + CR3_raw + Ensemble Learning	64.71% (+20.04%)

a lack of necessary contextual information to guide the repair.

Finding 7: Method-level BFPs and the limited model input/output lengths may miss the necessary contextual information to guide the repair, thus limiting the repair capability of LLMCs.

Computing Resource and Model Size. The lack of computing resources and the overly-large model size can hinder LLMCs from generating more candidate patches. For example, when we perform patch synthesis on Defects4J, we can only go up to a max beam size of 200 and generate 200 patches for each bug. Unlike traditional DL models [8], [12], [15], [50], it is non-trivial to use a larger beam size and produce a larger patch space. On the other hand, previous studies [7], [18] have confirmed that the size of patch space positively impacts the overall performance of DL models for APR. Therefore, such a limitation makes it difficult for further improvement.

Finding 8: The limited computing resource along with the large model size of LLMCs is a non-negligible factor that should be carefully considered when improving the fine-tuning of LLMCs.

V. Security Vulnerability Repair

A. Empirical Results

In Task ❺, we explore the vulnerability repair capability of LLMCs on the VulRepair dataset. Table III shows the repair results using our previously obtained best representation (*CR3*) and VRepair's representation (*CR4*) [41].

B. Research Questions

1) RQ1: **How do different design choices affect LLMCs' repair capability?** We conduct ablation study on CodeT5 used in VulRepair to explore the impact of different design choices.

Code Representation. As shown in Table VIII, the repair results using *CR3* are all better than those using *CR4*. Obviously, *CR3* is a more useful representation. This result again supports our **Finding 2**. This is because the two representations differ in the complexity of marking repair behaviors. As shown in Figure 2, in *CR3*, all repair behaviors are seen as replace operations. In *CR4*, three distinct marks are used to represent add, delete, and replace actions, providing finer token-level fix locations. However, the complexity of this strategy might impede the model in understanding the different repair actions and accurately implementing fixes at precise locations. As a result, the model's repair capability could be compromised. In fact, all repair actions can be simplified to replacement operations (i.e., the fix replaces the bug location).

TABLE IX: Best repair results for LLMCs on Task ❺

Our Work					Baseline (Fu et al. [18])	
CodeBERT	GraphCode.	PLBART	CodeT5	UniXcoder	VulRepair	VRepair
52.17%	54.16%	60.90%	64.71%	63.77%	44.67%	23.00%

TABLE X: Results of UniXcoder(-nine) on Task ❺

Details	UniXcoder	UniXcoder-nine
CR3_raw + PPL	50.64%	43.90%
CR3_raw + BLEU	62.08%	62.37%
CR3_raw + Last	62.19%	62.66%
CR3_raw + ALL	63.77%	64.71%

Although *CR3* uses a coarse-grained markup approach, it simplifies the repair operation, thereby enhancing the model's repair capability.

Finding 9: Using finer-grained code representations is not conducive to fully exploiting the repair capability of LLMCs, and CR3 remains the best representation on vulnerability repair.

Checkpoint Selection. As shown in Table VIII, the repair results using *BLEU* are better than those using *PPL* and *LOSS*. We infer this due to the random partitioning of the VulRepair dataset into train/val/test sets from Big-Vul and CVEfixes. Based on previous findings (Section IV-B1), *BLEU* is better suited for this scenario. After combining multiple models, we achieve up to 20.04% improvement in repair accuracy using CodeT5. This demonstrates that the ensemble learning strategy effectively improves the overall repair capability. These outcomes reinforce **Finding 3** in Section IV-B1.

2) RQ2: **How well does the LLMC perform compared to the state-of-the-art approaches?** Based on the findings obtained from RQ1, we use the data form *CR3_raw* and the ensemble learning strategy and compare them with baselines.

Performance. As shown in Table IX, on the VulRepair dataset, all LLMCs outperform VulRepair [18] and VRepair [41]. In particular, LLMCs improve over the best baseline VulRepair [18] as follows: CodeT5 (+20.04%) > UniXcoder (+19.10%) > PLBART (+16.23%) > GraphCodeBERT (+9.49%) > CodeBERT (+7.50%).

According to the results, we conclude that LLMCs can dramatically surpass the baselines [18], [41] on the vulnerability repair task. This also indicates that there is still a vast research space to improve vulnerability repair based on LLMCs.

Generalization. Importantly, several LLMCs (e.g., CodeBERT, GraphCodeBERT, PLBART, UniXcoder) not originally pre-trained in C/C++ language still demonstrate effective performance. The absence of data leak issues confirms their impressive generalization abilities. To further explore LLMCs' transferability, we compare UniXCoder with its C/C++ pre-trained variant UniXcoder-nine [55]. As shown in Table X, UniXcoder-nine has a slight improvement over UniXcoder. Nevertheless, the improvement is quite limited, suggesting that LLMC already has strong generalization capability.

Multi-Hunk. We also study the repair capability of LLMCs for different vulnerability hunks. As shown in Table XI, LLMCs have the highest repair accuracy in single-hunk fixes. In the multi-hunk fixing scenario, when the number of hunks is not big (i.e., <5), the accuracy is not significantly behind

TABLE XI: Performance of LLMCs with vulnerability hunks.

Vul. #Hunks	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder
1	60%	63%	71%	75%	74%
2	53%	55%	60%	64%	63%
3	56%	57%	62%	66%	65%
4	40%	42%	51%	54%	58%
5	25%	27%	31%	31%	33%
6	24%	24%	24%	27%	24%
7	17%	17%	22%	22%	17%
8	13%	13%	13%	13%	13%
9	7%	7%	7%	7%	7%
10+	5%	5%	5%	5%	5%

TABLE XII: Best repair results for LLMCs on Task ❻

Our Work					Baseline (Berabi et al. [25])		
CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	TFix	CoCoNuT	SequenceR
57.42%	56.45%	53.28%	60.62%	60.10%	49.30%	11.70%	17.90%

that of the single-hunk fixes. However, as the vulnerable hunks increase, the repair accuracy of LLMCs drops sharply. To sum up, although we can obtain a fairly good results for repairing complex vulnerabilities, there is still a long way to go for overly complex ones.

Finding 10: The fine-tuned LLMCs show great potential for vulnerability repair and have a strong generalization capability. They can also deal with multi-hunk vulnerabilities to some extent unless the hunks are too many.

3) RQ3: **What are the factors that limit the effectiveness of fine-tuning LLMCs?** Here we analyze the limitations that LLMC exhibits when dealing with different vulnerability types, input/output lengths, and vulnerability hunk numbers in the *VulRepair dataset*.

Long-tail Problem. Table XIII present the results of LLMC's repair capability on the top 10 CWE types [56]. When there is less training samples for one type of vulnerabilities, the performance is worse. This is the classic long-tail problem that challenges the effective training of a model on a large dataset with imbalanced class distribution. For example, the training set contains 97 CWE types, yet 55 types have no more than 10 training samples. In summary, the long-tail issue [39], [41] remains a major challenge that hinders vulnerability repair.

Finding 11: Addressing small sample sizes and class imbalance constitutes a primary challenge in vulnerability repair.

Long Sequence. We examine how the repair capability of LLMCs is influenced by the input and output lengths. From Table XIV, we observe that LLMCs suffer from the long sequence problem, i.e., as the length of input/output sequences increases, the repair capability of the model decreases. Even many studies [7], [10], [18] have highlighted the long sequence problem, alleviating this problem remains the way forward.

Finding 12: The repair capability of LLMCs suffers from the long sequence problem in general.

VI. Programming Error Repair

A. Empirical Results

In Task ❻, we explore the single-hunk error repair capability of LLMCs on the TFix dataset (see Table III).

B. Research Questions

1) RQ1: **How do different design choices affect LLMCs' repair capability?** Section IV and V have concluded that using the *CR3_raw* and the ensemble strategy is the best technical detail setup. Thus we will not explore RQ1 further.

```
  // fix no-undef Undefined variable.
  NL.triggerMapMoveEnd();
- NL.respondLast200(NL.json.MapWmsLayers.records.regular);
+ NL.respondLast200(fx.regular);
  var layers = NL.vw.MAP.getWmsLayers();
```

Fig. 5: An error-fix example of *no-undef*.

2) RQ2: **How well does the LLMC perform compared to the state-of-the-art approaches?** We use the *CR3_raw* and the ensemble strategy to obtain best repair results of LLMCs on test benchmark and compare them with baselines.

Task ❻. As shown in Table XII, on the TFix dataset, LLMCs improve over the best baseline TFix [25] (T5-large) as follows: CodeT5 (+11.32%) > UniXcoder (+10.80%) > CodeBERT (+8.12%) > GraphCodeBERT (+7.15%) > PLBART (+3.98%).

Overall, LLMCs have outperformed baseline approaches, underscoring the significant potential of LLMCs in error repair.

Finding 13: Fine-tuned LLMCs show significantly stronger repair capabilities compared to existing learning-based approaches.

3) RQ3: **What are the factors that limit the effectiveness of fine-tuning LLMCs?** Here we analyze the limitations that LLMC exhibits when dealing with different error types (see Table XV). We highlight the results of different error types where the accuracy is less than 50%.

Taking the *no-undef* error type as an example (see Figure 5), we observe that fixing this error requires sufficient context to substitute the undefined variable with the defined one. However, the TFix dataset solely furnishes context from the line preceding and following the error location. As a result, it might lack the essential repair components, potentially resulting in an incorrect patch. This issue re-emphasizes **Finding 7**.

VII. Discussion

This section discusses the limitations identified in our study that affect the repair capability of LLMCs and seeks directions for improvement.

1) Loss of Pre-trained Knowledge: As described in **Finding 6**, after fine-tuning, LLMCs may lose some of the knowledge learned from the pre-training phase compared to zero-shot learning [17]. Furthermore, we noticed that AlphaRepair [17] converts the repair task into a *cloze* task (MLM) rather than a *translation* task (NMT). The *cloze* task could better fit the model's pre-training task (i.e., MLM). That is, it predicts the token at the mask location based on the contextual tokens. However, it is unclear how the repair ability differs using the two paradigms (NMT and MLM). Therefore, we suggest exploring the following two directions.

D1: Mitigation of catastrophic forgetting. There have been various mitigation measures towards this problem [54], and it is meaningful to introduce these techniques into APR.
D2: NMT vs. MLM. Fine-tuning the LLMC through both NMT and MLM tasks allows us to explore the differences in repair capabilities between the two learning paradigms.

TABLE XIII: The % perfect predictions of LLMCs for the top-10 most dangerous CWEs. (T.S.: Train #Samples)

Vulnerability Type			CodeBERT		GraphCodeBERT		PLBART		CodeT5		UniXcoder	
Top	**CWE**	**T.S.**	**%PP**	**Proportion**	**%PP**	**Proportion**	**%PP**	**Proportion**	**%PP**	**Proportion**	**%PP**	**Proportion**
1	CWE-787	188	42%	26/62	42%	26/62	45%	28/62	52%	32/62	52%	31/62
2	CWE-79	7	100%	1/1	100%	1/1	100%	1/1	100%	1/1	100%	1/1
3	CWE-89	5	75%	3/4	75%	3/4	100%	4/4	100%	4/4	100%	4/4
4	CWE-20	455	62%	82/133	62%	83/133	69%	92/133	74%	98/133	74%	96/133
5	CWE-125	455	49%	81/165	51%	84/165	54%	89/165	57%	94/165	57%	93/165
6	CWE-78	21	0%	0/4	0%	0/4	0%	0/4	0%	0/4	0%	0/4
7	CWE-416	174	58%	37/64	58%	37/64	64%	41/64	67%	43/64	67%	43/64
8	CWE-22	23	50%	3/6	50%	3/6	50%	3/6	50%	3/6	50%	3/6
9	CWE-352	8	0%	0/1	0%	0/1	0%	0/1	0%	0/1	0%	0/1
10	CWE-434	1	0%	0/0	0%	0/0	0%	0/0	0%	0/0	0%	0/0
Total			**53%**	**233/440**	**54%**	**237/440**	**59%**	**258/440**	**63%**	**275/440**	**62%**	**271/440**

TABLE XIV: Impact of input/output lengths on LLMCs for vulnerability repair.

CodeBERT		**Input Length**					
		0-100	**101-200**	**201-300**	**301-400**	**401-500**	**500+**
Output Length	**0-10**	69%	65%	75%	65%	86%	63%
	11-20	53%	60%	75%	57%	80%	52%
	21-30	46%	57%	65%	57%	50%	65%
	31-40	44%	61%	67%	47%	59%	72%
	41-50	37%	59%	39%	50%	58%	44%
	50+	29%	28%	32%	27%	14%	24%

GraphCode		**Input Length**					
		0-100	**101-200**	**201-300**	**301-400**	**401-500**	**500+**
Output Length	**0-10**	76%	72%	77%	71%	76%	65%
	11-20	57%	64%	77%	57%	80%	52%
	21-30	49%	59%	65%	57%	50%	67%
	31-40	52%	61%	68%	53%	59%	74%
	41-50	37%	59%	39%	50%	58%	50%
	50+	32%	28%	33%	29%	18%	25%

PLBART		**Input Length**					
		0-100	**101-200**	**201-300**	**301-400**	**401-500**	**500+**
Output Length	**0-10**	77%	85%	82%	88%	86%	71%
	11-20	74%	74%	84%	64%	83%	56%
	21-30	78%	74%	70%	57%	50%	67%
	31-40	63%	70%	71%	53%	59%	70%
	41-50	63%	74%	39%	50%	58%	50%
	50+	54%	35%	41%	27%	18%	27%

CodeT5		**Input Length**					
		0-100	**101-200**	**201-300**	**301-400**	**401-500**	**500+**
Output Length	**0-10**	81%	89%	82%	94%	95%	72%
	11-20	81%	70%	88%	75%	83%	61%
	21-30	83%	81%	73%	71%	50%	68%
	31-40	63%	76%	79%	60%	59%	68%
	41-50	63%	74%	50%	70%	58%	58%
	50+	57%	37%	42%	29%	21%	21%

UniXcoder		**Input Length**					
		0-100	**101-200**	**201-300**	**301-400**	**401-500**	**500+**
Output Length	**0-10**	83%	83%	86%	88%	95%	72%
	11-20	82%	79%	88%	71%	83%	60%
	21-30	83%	74%	73%	63%	50%	71%
	31-40	63%	73%	79%	60%	59%	68%
	41-50	63%	74%	50%	60%	58%	58%
	50+	54%	32%	38%	31%	21%	27%

2) Lack of Repair Ingredients and Long Sequence Problem: As described in **Finding 7**, the input/output length limit of the model make LLMCs can not cover sufficient repair ingredients, which in turn constraints the repair capability. Furthermore, **Finding 12** also points out that LLMCs suffer from the long sequence problem.

D1: Precise context extraction. Through data/control flow analysis, we can trim irrelevant context [39], aid in pinpointing defect locations and guide repairs.
D2: Essential repair ingredients. We can integrate traditional APR techniques based on redundancy assumptions [52] with LLMCs to introduce additional repair elements into the model input.
D3: Breaking the length limit. One way to model long sequences for covering more repair ingredients is MegaByte [57]. In addition, adopting sliding-encoder and decoder (SLED) [26] to partition the input into overlapping chunks may also help accept long and/or dependent methods.

3) Computing Resource and Model Size: As described in **Finding 8**, generally the large model size raises high demands for computing resources. Therefore, how to optimize the deployment of LLMCs for application in low-resource scenarios is a practical problem.

D1: Model distillation. It optimizes existing model sizes while maintaining performance close to the original model [58].
D2: Efficient inference. Sparse mechanism [27] and limited discrepancy beam search [28] can also be applied when generating larger number of candidate patches.
D3: Parameter-efficient fine-tuning (PEFT). We can refine fine-tuning strategies by employing PEFT techniques, known for their efficiency and low-resource nature [59].

4) Long-tail Problem: As described in **Finding 11**, the long-tail problem caused by imbalanced type distribution in vulnerability repair tasks remains a key challenge. Also, there is the small sample problem. Previous works [39], [41] have proposed using transfer learning to alleviate this problem.

D1: Data augmentation. Designing data augmentation schemes [60] can expand the number of training samples and mitigate the challenge of small sample sizes.
D2: Combining multiple PLs. We can also use meta learning [61] that integrates multiple PLs to enhance the multilingual repair capability of LLMCs.

5) Multi-Hunk Fixes: As described in **Finding 10**, LLMCs still have difficulty dealing with complex multi-hunk fixes.

D1: Capturing complex code dependencies. Tree [24] or graph [62] structures that capture global dependencies can enhance the model's ability to understand and handle complex repair tasks.
D2: Extracting in-depth semantic information. Leveraging high-level semantic information (such as bytecode [63] and intermediate representation [64]) can aid the model in comprehending the root cause of defects, thereby enhancing its repair capability.

VIII. Threats to Validity

Internal. Existing approaches typically use different training datasets, patch space sizes, post-processing strategies, and other details, and it would be unfair to compare these APR tools directly [65]. To mitigate this threat, we used the same dataset and beam size as baselines. Note that when comparing with DEAR [24], their paper did not specify a specific patch space size, so we followed the practice of previous works [9], [42], [50] and chose a minimum beam size of 100 [9]. Also, our work did not use patch filtering and re-ranking strategies, whereas some baselines like DEAR adopted the post-processing for improvement. Therefore, our results could be further improved and our comparison is fair for baselines.

External. Although we have conducted a comprehensive study of 5 LLMCs for APR, with a variety of scenarios (e.g., 3 defect types, 7 test benchmarks, and 3 PLs), our results may still not generalize well to other LLMCs and PLs [66].

TABLE XV: Repair results of LLMCs across different error types.

Error Type	Sample	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder
no-new-symbol	1	100.0%	100.0%	100.0%	100.0%	100.0%
no-compare-neg-zero	2	0.0%	0.0%	0.0%	0.0%	0.0%
no-ex-assign	4	50.0%	50.0%	50.0%	50.0%	50.0%
for-direction	5	40.0%	60.0%	20.0%	80.0%	80.0%
no-unsafe-finally	7	42.9%	42.9%	42.9%	42.9%	57.1%
use-isnan	8	37.5%	37.5%	25.0%	50.0%	50.0%
no-dupe-class-members	12	0.0%	0.0%	0.0%	8.3%	8.3%
no-class-assign	12	66.7%	58.3%	58.3%	75.0%	66.7%
no-func-assign	15	40.0%	60.0%	53.3%	60.0%	53.3%
no-empty-pattern	18	38.9%	50.0%	38.9%	44.4%	50.0%
no-unused-labels	19	57.9%	63.2%	52.6%	57.9%	63.2%
no-duplicate-case	20	65.0%	65.0%	60.0%	65.0%	65.0%
getter-return	21	61.9%	61.9%	61.9%	57.1%	52.4%
no-sparse-arrays	24	33.3%	41.7%	45.8%	50.0%	45.8%
no-const-assign	28	32.1%	28.6%	39.3%	42.9%	35.7%
no-global-assign	32	78.1%	78.1%	65.6%	81.2%	78.1%
no-new-wrappers	36	38.9%	41.7%	41.7%	55.6%	52.8%
no-this-before-super	42	76.2%	83.3%	69.0%	85.7%	83.3%
no-unsafe-negation	43	67.4%	76.7%	76.7%	79.1%	79.1%
require-yield	43	79.1%	72.1%	81.4%	83.7%	76.7%
no-new-object	45	64.4%	64.4%	62.2%	68.9%	68.9%
no-caller	45	22.2%	22.2%	22.2%	31.1%	28.9%
no-extend-native	45	40.0%	46.7%	33.3%	51.1%	46.7%
constructor-super	47	80.9%	80.9%	80.9%	80.9%	78.7%
valid-typeof	54	57.4%	55.6%	40.7%	63.0%	57.4%
no-self-assign	61	50.8%	50.8%	45.9%	54.1%	52.5%

Error Type	Sample	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder
no-extra-bind	69	73.9%	72.5%	73.9%	75.4%	75.4%
no-case-declarations	73	68.5%	67.1%	67.1%	68.5%	71.2%
no-fallthrough	75	78.7%	76.0%	76.0%	78.7%	80.0%
no-inner-declarations	84	53.6%	56.0%	46.4%	59.5%	56.0%
no-array-constructor	98	86.7%	86.7%	85.7%	86.7%	84.7%
no-constant-condition	129	54.3%	49.6%	48.1%	58.9%	52.7%
generator-star-spacing	140	72.1%	70.0%	69.3%	75.0%	77.1%
no-extra-boolean-cast	146	61.6%	61.6%	50.7%	61.0%	64.4%
no-cond-assign	152	53.9%	50.0%	52.6%	56.6%	54.6%
no-process-exit	152	43.4%	42.8%	38.8%	44.1%	43.4%
no-empty	207	41.1%	41.1%	37.7%	40.1%	45.4%
no-dupe-keys	219	60.3%	60.3%	58.0%	63.5%	63.0%
prefer-spread	250	56.4%	52.0%	46.4%	55.6%	54.0%
no-useless-escape	293	39.2%	36.5%	33.1%	47.4%	45.1%
no-console	307	77.2%	78.2%	77.2%	77.2%	79.2%
guard-for-in	324	48.8%	47.5%	38.6%	54.3%	52.8%
no-throw-literal	408	71.8%	68.1%	69.1%	77.0%	75.7%
no-debugger	417	96.4%	96.9%	95.9%	96.9%	96.2%
prefer-rest-params	459	42.3%	42.5%	35.7%	47.3%	47.5%
no-unreachable	473	72.1%	70.8%	70.2%	74.2%	74.0%
no-extra-semi	598	85.8%	84.4%	85.8%	86.5%	86.8%
no-redeclare	639	57.9%	58.4%	57.0%	62.6%	61.7%
comma-style	640	60.9%	55.5%	52.7%	63.7%	61.3%
no-unused-vars	777	64.2%	62.3%	59.2%	65.1%	65.9%
no-undef	1064	31.5%	30.5%	25.2%	35.7%	35.0%
no-invalid-this	1622	46.2%	46.5%	42.1%	50.1%	50.0%

For example, we did not include extremely LLMs (e.g., GPT-NeoX-20B) for APR, mainly because of the limited computing resource. However, we believe our results are representative for a relatively wide range of conditions. We will enhance it with more advanced resources and expect researchers of following work could improve our study.

IX. Related Work

A. LLMC-based APR

Mashhadi et al. [67] first used CodeBERT fine-tuning to solve single-line bug repair problems. Later, Huang et al. [68] investigated the repair effectiveness of using CodeBERT and GraphCodeBERT based on fine-tuning. Recently, Xia et al. [17] introduced *cloze* tasks into the APR domain, which uses LLMCs to predict the correct code for defect locations with context. They proposed the APR tool AlphaRepair based on CodeBERT and zero-shot learning, which provides a new direction towards APR. In addition, Xia et al. [52] proposed another novel APR tool, FitRepair, based on CodeT5, which combines the *plastic surgery hypothesis* with LLMCs to provide additional repair ingredients and thus enhance the repair capability of LLMCs. With the recent popularity of ChatGPT [69], Xia et al. [70] proposed ChatRepair, a conversational APR tool that provides a new workflow using LLMs by continuously learning knowledge and receiving feedback to enhance repair capability.

B. Study on LLMC for APR

Fan et al. [21] investigated whether APR techniques can improve the reliability of code produced by LLMCs (Codex) and provide suggestions for enhancing APR with the help of LLMCs. Xia et al. [16] and Pearce et al. [23] comprehensively explored the performance of LLMCs in bug and vulnerability repair tasks using the zero/few-shot learning paradigm. Besides, some studies [30], [31] systematically compared LLMCs on various tasks, which partially include APR. Unlike their work, we focus on the repair capabilities of LLMCs under the NMT fine-tuning paradigm. One similar work is Jiang et al. [22], which also used the fine-tuning paradigm. However, they focused on the comparison between zero-shot and fine-tuning for APR, and only explored the single-hunk Java bug repair task. In contrast, we make a comprehensive exploration of fine-tuning LLMCs for APR across multiple languages, various defect types, and different levels of bug/vulnerability complexity. We also provide guidance on selecting the appropriate designs to enhance the repair capabilities of LLMCs, and achieve the new SOTA results.

X. Conclusion

This paper conducts a comprehensive study on the repair capabilities of LLMCs in various repair scenarios under the NMT fine-tuning paradigm. Our results show that even without any post-processing strategies, LLMCs can already achieve excellent results, and surpass many previous APR works. Importantly, we present some practical guidelines on how to choose different designs to better exploit the repair capability of LLMCs, and show how they can repair complex defects. We also analyze and discuss some limitations found during the evaluation and point out future directions. Furthermore, our results on various benchmarks can serve as the baselines for subsequent works with reference. In conclusion, LLMC-based APR has great potential for practical use, and more efforts are needed to promote LLM4APR research in the future.

Acknowledgment

We thank the anonymous reviewers for their insightful comments to improve this paper. This work was partially supported by the National Key R&D Project (2023QY1202), National Natural Science Foundation of China (U1836210), and Hainan Key R&D Project (GHYF2022010). This work was also partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, 2018.

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Software Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[3] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[4] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," *arXiv preprint arXiv:2211.12787*, 2022.

[5] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[6] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," *arXiv preprint arXiv:2303.18184*, vol. abs/2303.18184, 2023.

[7] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[8] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd International Conference on Software Engineering, ICSE*, 2021, pp. 1161–1173.

[9] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2021, pp. 341–353.

[10] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1943–1959, 2021.

[11] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE*, 2020, pp. 602–614.

[12] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, 2020, pp. 101–114.

[13] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *Proceedings of the 37th International Conference on Machine Learning, ICML*, vol. 119, 2020, pp. 10 799–10 808.

[14] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE*, 2022, pp. 1506–1518.

[15] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023, pp. 1456–1468.

[16] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023, pp. 1482–1494.

[17] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2022, pp. 959–971.

[18] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Q. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2022, pp. 935–947.

[19] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, "Repair is nearly generation: Multilingual program repair with llms," *arXiv preprint arXiv:2208.11640*, 2022.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP*, ser. ACL, 2020, pp. 1536–1547.

[21] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," *arXiv preprint arXiv:2205.10583*, 2022.

[22] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023, p. 1430–1442.

[23] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy, SP*, 2022, pp. 1–18.

[24] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering, ICSE*, 2022, pp. 511–523.

[25] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning, ICML*, 2021, pp. 780–791.

[26] M. Ivgi, U. Shaham, and J. Berant, "Efficient long-text understanding with short-text models," *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 284–299, 2023.

[27] S. Jaszczur, A. Chowdhery, A. Mohiuddin, L. Kaiser, W. Gajewski, H. Michalewski, and J. Kanerva, "Sparse is enough in scaling transformers," *Advances in Neural Information Processing Systems, NIPS*, vol. 34, pp. 9895–9907, 2021.

[28] D. Furcy and S. Koenig, "Limited discrepancy beam search," in *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI*, 2005, pp. 125–131.

[29] "Llmc4apr study." [Online]. Available: https://github.com/LLMC-APR/STUDY

[30] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st International Symposium on Software Testing and Analysis, ISSTA*, 2022, pp. 39–51.

[31] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023, pp. 2136–2148.

[32] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations, ICLR*, 2021.

[33] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," *arXiv preprint arXiv:2301.09072*, 2023.

[34] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[35] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2020, pp. 1433–1443.

[36] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[37] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the Empirical Methods in Natural Language Processing, EMNLP*, 2021, pp. 8696–8708.

[38] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association*

for Computational Linguistics: Human Language Technologies, NAACL-HLT, 2021, pp. 2655–2668.

[39] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "Seqtrans: Automatic vulnerability fix via sequence to sequence learning," *IEEE Trans. Software Eng.*, vol. 49, no. 2, pp. 564–585, 2023.

[40] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," in *Proceedings of the 37th International Conference on Automated Software Engineering, ASE*, 2022, pp. 1–13.

[41] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in C code," *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 147–165, 2023.

[42] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: continual repair across programming languages," in *Proceedings of the 31st International Symposium on Software Testing and Analysis, ISSTA*, 2022, pp. 678–690.

[43] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *arXiv preprint arXiv:2208.03133*, 2022.

[44] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th International Conference on Automated Software Engineering, ASE*, 2022, pp. 1–13.

[45] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering, ICSE*, 2022, pp. 1169–1180.

[46] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, ACL*, 2022, pp. 7212–7225.

[47] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA*, 2014, pp. 437–440.

[48] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories, MSR*, 2020, pp. 508–512.

[49] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE*, 2021, pp. 30–39.

[50] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023, pp. 1251–1263.

[51] Z. Chen and M. Monperrus, "The codrep machine learning on source code competition," *arXiv preprint arXiv:1807.03200*, 2018.

[52] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," *arXiv preprint arXiv:2303.10494*, 2023.

[53] B. Baudry, Z. Chen, K. Etemadi, H. Fu, D. Ginelli, S. Kommrusch, M. Martinez, M. Monperrus, J. R. Arteaga, H. Ye, and Z. Yu, "A software-repair robot based on continual learning," *IEEE Softw.*, vol. 38, no. 4, pp. 28–35, 2021.

[54] C. Shao and Y. Feng, "Overcoming catastrophic forgetting beyond continual learning: Balanced training for neural machine translation," *arXiv preprint arXiv:2203.03910*, 2022.

[55] "Unixcoder-base-nine." [Online]. Available: https://huggingface.co/microsoft/unixcoder-base-nine

[56] "2022 cwe top 25 most dangerous software weaknesses," 2022. [Online]. Available: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[57] L. Yu, D. Simig, C. Flaherty, A. Aghajanyan, L. Zettlemoyer, and M. Lewis, "Megabyte: Predicting million-byte sequences with multiscale transformers," 2023.

[58] K. J. Liang, W. Hao, D. Shen, Y. Zhou, W. Chen, C. Chen, and L. Carin, "Mixkd: Towards efficient distillation of large-scale language models," in *Proceedings of the 9th International Conference on Learning Representations, ICLR*, 2021.

[59] M. Sourab, G. Sylvain, D. Lysandre, B. Younes, and P. Sayak, "Peft: State-of-the-art parameter-efficient fine-tuning methods," 2022. [Online]. Available: https://github.com/huggingface/peft

[60] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *Proceedings of the 45th International Conference on Software Engineering, ICSE*, 2023.

[61] C. Park, Y. Tae, T. Kim, S. Yang, M. A. Khan, L. Park, and J. Choo, "Unsupervised neural machine translation for low-resource domains via meta-learning," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP*, 2021, pp. 2888–2901.

[62] S. Liu, X. Xie, J. K. Siow, L. Ma, G. Meng, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2839–2855, 2023.

[63] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th International Symposium on Software Testing and Analysis, ISSTA*, 2019, pp. 19–30.

[64] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering, ICSE*, 2022, pp. 2253–2265.

[65] W. Zhong, H. Ge, H. Ai, C. Li, K. Liu, J. Ge, and B. Luo, "Standup4npr: Standardizing setup for empirically comparing neural program repair systems," in *Proceedings of the 37th International Conference on Automated Software Engineering, ASE*, 2022, pp. 1–13.

[66] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2020, p. 1556–1560.

[67] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proceedings of the 18th International Conference on Mining Software Repositories, MSR*, 2021, pp. 505–509.

[68] K. Huang, S. Yang, H. Sun, C. Sun, X. Li, and Y. Zhang, "Repairing security vulnerabilities using pre-trained programming language models," in *Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W*, 2022, pp. 111–116.

[69] "Introducing chatgpt," 2022. [Online]. Available: https://openai.com/blog/chatgpt

[70] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.